Project 2: Goldchase–*verbis ad verbera*

This project enhances the Goldchase game you coded for the first project. There are two features which will be added to improve the game: automatic screen updates and messages between players.

Game play enhancements

This version of Goldchase will include the following enhancements:

1. When a new player enters a game which has already started, the other players' maps will automatically update to show the new player's position on the board.
2. When any player changes the map (such as, for example moving), then the other players' maps will automatically update.
3. Type 'm' to send a message to another player. Prompt the user with a list of active players from which one should be selected (see `getPlayer()` below). The user will then enter a message (see `getMessage()` below) which will appear on the respective player's window in the form of a `postNotice`. Be sure to prepend the message with "`Player # says:`".
4. Type 'b' to broadcast a message to all other players. Prompt the user to enter a message. This message will be broadcast to all other players as `postNotice` messages on their respective windows. Be sure to prepend the message with "`Player # says:`".
5. When a player wins by leaving the game board with the gold, a "`Player # won!`" message will be sent to the other players.
6. As before, the last person/process to leave the game is responsible for unlinking the shared memory and semaphore.
7. If a process ends abnormally (e.g., SIGTERM, SIGINT, SIGHUP) be sure to unlink any message queues belonging to the process. Also, unlink shared memory and the semaphore if it is the last process.

Architectural considerations

Signal handling and message queues will be used to implement automatic screen updates and chat functionality.

1. The Map class has two additional functions:
   (1) `int getPlayer(int PlayerMask)` This member function accepts a single integer which is some combination of the `G_PLR#` bits. A box is displayed showing a player number matching each bit. As soon as a valid player number is typed in by the user, the box disappears and the entered `G_PLR#` corresponding to the selected player is returned. Usage example: Let's say that yourself, player 5, and player 2 are playing the game. If you want to send a message to one of these players, then an unsigned integer needs to be initialized to `(G_PLR4 | G_PLR1)` and passed to the getPlayer function. A box with 5 and 2 will appear on the screen. Once the user types a '2' or a '5', either `G_PLR1` or `G_PLR4` will be returned by the function.

(2) `std::string getMessage()` This member function prompts the user for a string of text, which is returned. Use this function to query the player for the text that should be sent to a different player.

2. The mapBoard structure will need to be expanded. It's original form included the following fields:

   ```
   struct mapBoard {
     unsigned int columns;
     unsigned int rows;
     unsigned char players;
     char* board[0];
   };
   ```

   Expand this structure with five more integers–one for each of the possible five players (type `pid_t` would be more correct than integers). A single array of 5 elements would work well. If a player is not playing (or leaves the game), the respective integer should be set to zero. If a player joins the game, the integer should be set to the process ID of that player's process. When you need to send a signal to the other players (such as when you call drawMap), you can examine these 5 integers to determine the process IDs to send signals to.
3. The best way to implement the chat functionality is to have a separate message queue for each player. The "owner" of the queue will only read from the queue, all other players will only write to the queue.

Additional message queue notes

Message queues can register for signals. That is, your program can inform a message queue that it wants to receive a signal when a message is sent to the (empty) queue. Here is some sample code which shows how a program would register to receive the `SIGUSR2` signal when a message hits the empty queue:

```
int mqfd=mq_open(...,O_RDONLY|...|O_NONBLOCK, ...); //init mqfd
...
struct sigevent mq_notification_event; //create a structure object
mq_notification_event.sigev_notify=SIGEV_SIGNAL; //signal notification
mq_notification_event.sigev_signo=SIGUSR2; //using the SIGUSR2 signal
mq_notify(mqfd, &mq_notification_event); //register it!
```

Note that this is a single-use trigger. Once the signal has been sent, it will not be sent again unless this setup is done again. Therefore, the code above needs to be repeated in the signal handler in order to re-register for signal notification. Also note that the `O_NONBLOCK` flag has been added to the various open flags. This simply makes our `mq_receive()` calls non-blocking.

The signal handler for message queues should be structured like this:

```
void handle_message_receipt(int)
{
   //re-register for signals via the struct sigevent stuff

   char msg[250];  //a char array for the message
   memset(msg,0,250);  //zero it out (if necessary)
   while(mq_receive(mqfd, msg, 250, NULL)!=-1)
     //call postNotice(msg) for your Map object;
   if(errno!=EAGAIN) //EAGAIN expected after all messages are read
   {
     perror("mq_receive error"); //if errno != EAGAIN
     exit(1);
   }
}
```